Cálculo Lambda em Haskell

Rodrigo Machado

2 de junho de 2011

Este documento reporta os passos necessários para a construção de um interpretador com interface gráfica para cálculo lambda usando a linguagem Haskell.

1 Sintaxe

Definição do módulo principal e bibliotecas utilizadas.

```
module Main where
import Data.List
import Data.IORef
import Graphics.UI.Gtk
import Graphics.UI.Gtk.Glade
import Text.ParserCombinators.Parsec
```

O primeiro passo é descrever um tipo de dados para representar termos lambda (sintaxe abstrata). Para variáveis, nós utilizaremos Id como um sinônimo para o tipo pré-definido String.

```
type Id = String
data Term = Var Id
          | Lambda Id Term
          | App Term Term
          deriving (Eq,Ord,Show,Read)
```

2 Ocorrência de variáveis

Esta seção define funções que lidam com os conceitos de variável livre e ligada, como segue:

- vars: extrai todas as variáveis que ocorrem no termo.

```
vars :: Term → [Id]
vars (Var x)      = [x]
vars (Lambda x t) = [x] `union` (vars t)
vars (App t1 t2)  = (vars t1) `union` (vars t2)
```

- fv: extrai todas as variáveis que ocorrem livres no termo.

```
fv :: Term → [Id]
```

```
fv (Var x)      = [x]
fv (Lambda x t) = (fv t) \\ [x]
fv (App t1 t2)  = (fv t1) `union` (fv t2)
```

3 Substituição

Agora definimos substituição de variáveis. A chamada (sub x v t) pode ser lido como $[x := v](t)$, isto é, substituir em t todas as ocorrências livres da variável x pelo termo v.

```
sub :: Id → Term → Term → Term
sub x v (Var y)       | y == x    = v
                      | otherwise = Var y
sub x v (Lambda y t)  | y == x    = Lambda y t
                      | otherwise = Lambda y (sub x v t)
sub x v (App t1 t2)               = App (sub x v t1) (sub x v t2)
```

A função sub, contudo, não evita a possível captura de variáveis livres em v. Por exemplo:

$$\mathsf{sub}\ y\ x\ (\lambda x.x\ y) = \lambda x.x\ x$$

Na posição da variável livre y (em $\lambda x.x\ y$), o nome x está ligado. Ao realizar a substituição, a variável livre x (segundo parâmetro) acaba se ligando, isto é se confundindo com o parâmetro formal x (em $\lambda x.x\ y$). Para evitar essa situação, é necessário poder realizar a troca do nome de variáveis ligadas.

4 Redução alfa

A troca de nomes de variáveis ligadas é obtida em cálculo lambda através da operação de redução alfa. Na implementação em Haskell, a função alpha recebe uma lista de variáveis que não podem ser usadas como variaveis ligadas e um termo.

```
alpha :: [Id] → Term → Term
alpha xs (Var y)                    = Var y
alpha xs (Lambda y t) | y `elem` xs = let n = newName xs in
                                        Lambda n (sub y (Var n) (alpha (n:xs) t))
                      | otherwise   = Lambda y (alpha xs t)
alpha xs (App t1 t2)                = App (alpha xs t1) (alpha xs t2)
```

A lista infinita names contém strings que podem ser usadas como variáveis. A função newName, chamada por alpha, cria novos nomes para ligações no termo, certificando-se que eles não ocorrem na lista de nomes recebida. Para tal, ela retorna o primeiro elemento de names que não esteja na lista recebida.

```
names :: [Id]
names = tail $ gen [[]]
        where  gen x = x ++ gen [ c:s | c ← ['a'..'z'], s← x ]

newName :: [Id] → Id
newName xs = head $ filter (`notElem` xs) names
```

Se nos certificarmos que não haverá nenhuma variável ligada com mesmo nome que uma variável livre, a operação de substituição se torna segura. Por exemplo:

$$\mathsf{alpha}\ [x]\ (\lambda x.x\ y) = (\lambda a.a\ y)$$
$$\mathsf{sub}\ y\ x\ (\lambda a.a\ y) = (\lambda a.a\ x)$$

5 Formas normais

Um *redex* ("reducible expression") é um termo lambda na forma $(\lambda x.M)N$. Um termo lambda é uma *forma normal* se não for um redex e não possuir nenhum redex como subtermo. A chamada (nf t) testa se t é uma forma normal.

```
nf :: Term → Bool
nf (Var _)              = True
nf (Lambda x t)         = nf t
nf (App (Lambda _ _) _) = False
nf (App t1 t2)          = (nf t1) && (nf t2)
```

6 Redução beta

Utilizando as funções alpha e sub, podemos finalmente definir a função de redução beta. O tipo de retorno de beta é Maybe Term pois é possível que o termo a ser avaliado seja uma forma normal (irredutível). Note que beta utiliza uma estratégia preguiçosa de avaliação, isto é, redução sempre pelo redex mais externo.

```
beta :: Term → Maybe Term
beta (Var x)                    = Nothing
beta (Lambda x t)               = do t' ← beta t
                                     return (Lambda x t')
beta (App (Lambda x t) t2)      = Just (sub x t2 (alpha (fv t2) t))
beta (App t1 t2) | not (nf t1)  = do t1' ← beta t1
                                     return (App t1' t2)
                 | otherwise    = do t2' ← beta t2
                                     return (App t1 t2')
```

Podemos também definir uma versão estrita de avaliação. Função betaStrict avalia a função e os argumentos até a formal normal (se houver) antes de reduzir o redex.

```
betaStrict :: Term → Maybe Term
betaStrict (Var x)                   = Nothing
betaStrict (Lambda x t)              = do t' ← betaStrict t
                                          Just (Lambda x t')
betaStrict (App t1 t2) | not (nf t1) = do t1' ← betaStrict t1
                                          Just (App t1' t2 )
                       | not (nf t2) = do t2' ← betaStrict t2
                                          Just (App t1  t2')
                       | otherwise   = case t1 of
                                         Lambda x t → Just (sub x t2
                                                            (alpha (fv t2) t))
                                         otherwise  → Nothing
```

7 Impressão legível e parser

Esta seção define funções de tradução entre a sintaxe concreta de termos lambda e a sintaxe abstrata. A primeira função (pretty) permite a visualização mais intuitiva de termos, convertendo-os da sintaxe abstrata para a sintaxe concreta.

```
pretty :: Term → String
pretty (Var s)                  = s
pretty (Lambda s t)             = "\\" ++ s ++ "." ++ pretty t
pretty (App t1@(Lambda _ _)
            t2@(Lambda _ _))    = "(" ++ pretty t1 ++ ") (" ++ pretty t2 ++ ")"
pretty (App t1@(Lambda _ _) t2) = "(" ++ pretty t1 ++ ") " ++ pretty t2
pretty (App t1 t2@(Lambda _ _)) = pretty t1 ++ " (" ++ pretty t2 ++ ")"
pretty (App t1 t2@(App _ _))    = pretty t1 ++ " (" ++ pretty t2 ++ ")"
pretty (App t1 t2)              = pretty t1 ++ " " ++ pretty t2
```

A outra função (pparser) é um parser de termos lambda. , construído através da biblioteca Parsec. A sintaxe concreta está disponível visualmente na interface gráfica. Note que o parser lê os termos lambda seguindo as convenções sintáticas habituais: aplicação é associativa à esquerda, maior escopo possível para abstração lambda.

```
termP :: Parser Term
termP =  do spaces
            (h:t) ← sepBy1 (lambP <|> varP <|> numP <|> parenP ) spaces
            spaces
            return (foldl App h t) -- aplicacao

lambP =  do char '\\'
            spaces
            x ← idP
            spaces
            char '.'
            spaces
            t ← termP
            spaces
            return (Lambda x t) -- lambda

varP =   do x ← idP
            spaces
            return (Var x)  -- variaveis

numP =   do xs ← many1 digit
            spaces
            return (cn (read xs))

parenP = do char '('
            t ← termP
            char ')'
            spaces
            return t -- parenteses

idP :: Parser String
idP = do x ←      (letter <|> char '_')
         y ← many (letter <|> digit <|> char '_')
```

```
        return (x:y) -- identificadores
```

Por conveniência, o parser termP aceita que o usuário escreva número naturais na posição de um termo. Esses números são codificados como termos lambda pela função cn definida a seguir.

```
-- Funcao que constroi o numeral de Church a partir de um inteiro
cn :: Integer → Term
cn n = Lambda "f" (Lambda "x" (rec n))
     where rec n | n ≤ 0    = (Var "x")
                 | otherwise = (App (Var "f") (rec (n-1)))
```

Outra conveniência é a apresentação de termos lambda como numerais de Church. Para tal, a função numRep "decodifica" um numeral de Church, e o apresenta como um número natural. Caso o termo seja mal-formado, então Nothing é retornado.

```
-- Funcao que retorna o número representado pelo termo, ou Nothing
-- se o termo for mal-formado
numRep :: Term → Maybe Integer
numRep (Var _)   = Nothing
numRep (App _ _) = Nothing
numRep (Lambda a (Lambda b t)) = collect 0 a b t
  where
    collect x i j (App (Var k) u) | i == k    = collect (x+1) i j u
                                  | otherwise = Nothing
    collect x i j (Var k)         | j == k    = Just x
                                  | otherwise = Nothing
    collect x i j _                           = Nothing
numRep (Lambda _ _) = Nothing
```

Programas são sequências de definições finalizadas por um termo. O parser progP lê programas inteiros, construindo um termo lambda único ao final do processo.

```
progP :: Parser Term
progP = do spaces
           (<|>) (do s ← idP
                     spaces
                     char '='
                     spaces
                     t ← termP
                     spaces
                     char ';'
                     u ← progP
                     return (App (Lambda s u) t))
                 (do string ">>"
                     u ← termP
                     return u)
```

8 Função principal e interface

A função principal simplesmente carrega a interface gráfica, desenvolvida com a ferramenta Glade e carregada dinamicamente a cada execução. Isto requer que o arquivo `lambda.glade` esteja no mesmo diretório do executável no momento da invocação.

```
main = do

  -- Inicializa Gtk
  initGUI

  -- Carrega interface
  Just xml ← xmlNew "./lambda.glade"

  -- Acessa objetos de interesse
  window   ← xmlGetWidget xml castToWindow     "window1"
  text1    ← xmlGetWidget xml castToTextView   "textview1"
  text2    ← xmlGetWidget xml castToTextView   "textview2"
  button1  ← xmlGetWidget xml castToButton     "button1"
  button2  ← xmlGetWidget xml castToButton     "button2"
  button3  ← xmlGetWidget xml castToButton     "button3"
  button6  ← xmlGetWidget xml castToButton     "button6"
  spin1    ← xmlGetWidget xml castToSpinButton "spinbutton1"
  label3   ← xmlGetWidget xml castToLabel      "label3"
  radio1   ← xmlGetWidget xml castToRadioButton "radiobutton1"
  radio2   ← xmlGetWidget xml castToRadioButton "radiobutton2"
  buf1     ← get text1 textViewBuffer
  buf2     ← get text2 textViewBuffer

  -- Define estado (termo atual, termos anteriores, passo, passo maximo, avalicao)
  state  ← newIORef ((   Nothing,     [],       0,       0, True) ::
                     (Maybe Term, [Term], Integer, Integer, Bool))

  -- Respostas a eventos
  onDestroy window  $ mainQuit                                          -- fecha janela
  onPressed button1 $ loadProgram buf1 buf2 label3 radio1 spin1 state -- carrega programa
  onPressed button2 $ stepBack         buf2 label3              state -- passo para tras
  onPressed button3 $ stepForward      buf2 label3              state -- passo para frente
  onPressed button6 $ runForward       buf2 label3              state -- avalia programa
  onToggled radio1  $ switchStrategy   radio1                   state -- muda avaliacao
  onToggled radio2  $ switchStrategy   radio1                   state -- muda avaliacao

  -- Exibe janela e executa loop principal
  widgetShowAll window
  mainGUI
```

Os tratadores de eventos são descritos a seguir.

```
loadProgram buf1 buf2 label3 radio1 spin1 state = do
  s1  ← get buf1   textBufferText
  ev  ← get radio1 toggleButtonActive
  mv  ← get spin1  spinButtonValue
  case (parse progP "" s1) of
       Left  _ → writeIORef state  (Nothing, [], 0,  0, True)
```

```
      Right t → writeIORef state  (Just t,  [], 0, floor mv,   ev)
  updateGUI   buf2 label3 state


stepBack buf2 label3 state = do
  (a,b,c,m,e) ← readIORef state
  case (a,b) of
    (Just t, h:d) → writeIORef state (Just h,d,c-1,m,e)
    _             → return ()
  updateGUI  buf2  label3  state


stepForward buf2 label3 state = do
  (a,b,c,m,e) ← readIORef state
  let ev        = if e then beta else betaStrict
  case do { t←a; h←ev t; return (t,h, c < m) }  of
    Just(t,h,True)  → writeIORef state (Just h,t:b,c+1,m,e)
    _               → return ()
  updateGUI buf2 label3 state


runForward buf2 label3 state = do
  (a,b,c,m,e) ← readIORef state
  case do {t←a; return (t, not (nf t), c < m)} of
    Just (t,True,True)  → do stepForward buf2 label3 state
                             runForward  buf2 label3 state
    _                   → updateGUI buf2 label3 state


runForward2 buf2 label3 state = do
  (a,b,c,m,e) ← readIORef state
  case do {t←a; return (t, not (nf t), c < m)} of
    Just (t,True,True)  → do stepForward buf2 label3 state
                             runForward  buf2 label3 state
    _                   → updateGUI buf2 label3 state

updateGUI buf2 label3 state = do
  (a,b,c,m,e) ← readIORef state
  case a of
    Just t  → do set buf2   [textBufferText := pretty t]
                 set label3 [labelText := l1 ++ l2 ++"Passo "++
                                          (show c)++"/"++show m]
                 where l1  = case numRep t of
                               Nothing → ""
                               Just x  → "(CN " ++ (show x) ++ ")    "
                       l2  = if e then "Avaliação Lazy    "
                                  else "Avaliação Strict    "
    Nothing → do set buf2   [textBufferText := "Programa mal-formado"]
                 set label3 [labelText      := ""]

switchStrategy r1 state = do
  (a,b,c,m,e) ← readIORef state
  isLazy      ← get r1 toggleButtonActive
  writeIORef state  (a,b,c,m,isLazy)
```

9 Construções em Lambda Cálculo

Esta seção apresenta algumas codificações comuns de tipos de dados como termos lambda. A sintaxe das definições pode ser carregada diretamente no interpretador.

```
[Valores verdade]
true  = \a. \b. a ;
false = \a. \b. b ;

[Condicional (if c then a else b)]
if    = \c. \a. \b. c a b ;

[Operadores booleanos]
and  = \a. \b. a b a ;
or   = \a. \b. a a b ;
not  = \p. \a. \b. p b a ;

[Construtor de pares]
pair = \a. \b. \c. c a b ;

[Operações de pares]
fst = \p. p true ;
snd = \p. p false ;

[Construtores de listas]
empty = pair true true ;
cons  = \h. \t. pair (pair false h) t ;

[Operações de listas]
isEmpty = \l. fst (fst l) ;
head = \l. snd (fst l) ;
tail = \l. snd l ;

[Números naturais]
0 = \f. \x. x ;
1 = \f. \x. f x ;
2 = \f. \x. f (f x) ;
3 = \f. \x. f (f (f x)) ;
 ...

[Operações numéricas]
succ = \n. \p. \q. p (n p q) ;
add  = \m. \n. \p. \q. (m p) (n p q) ;
mult = \m. \n. \p. \q.  m (add n) 0 ;
shiftInc = \p. pair (snd p) (succ (snd p)) ;
pred     = \n. fst (n shiftInc (pair 0 0)) ;
sub      = \m. \n. \p. \q. n pred (m p q) ;

[Operações relacionais]
isZero  = \n. n (\x. false) true ;

[Combinador de ponto fixo]
fix = \f. (\x. f (x x)) (\x. f (x x)) ;
```

10 Melhorias planejadas

- Comentários na sintaxe de entrada.
- Abrir arquivo e salvar arquivo.